



Università degli Studi di Parma

Dipartimento di Ingegneria e Architettura

Sistemi operativi e in tempo reale - a.a. 2022/23

Processi concorrenti e mutua esclusione



Il problema delle interferenze

- ❑ Interferenze: interazioni tra processi o thread non previste e non desiderate
- ❑ Errori di programmazione:
 - I tipo: inserimento di *interazioni tra thread o processi non necessarie*, non richieste dal problema da risolvere
 - Il tipo: *risoluzione erranea di interazioni tra thread o processi* comunque necessarie (per cooperazione o competizione)
- ❑ Per prevenire interferenze del primo tipo:
 - Incapsulamento della esecuzione all'interno dello spazio protetto di un *processo* (-> processo in stile Unix)
 - Non si possono verificare interferenze tra processi in spazi protetti
 - Attenzione: anche processi *indipendenti* presentano condivisione occulta di risorse ed interazione entro il kernel

Il problema delle interferenze



- ❑ In presenza di cooperazione o competizione (tipicamente tra *thread*) non ci sono meccanismi di protezione di sistema ed è necessario programmare in modo disciplinato e strutturato
- ❑ Il manifestarsi in modo *dipendente dal tempo* degli effetti di errori di programmazione concorrente ne rende problematica la rilevazione in fase di debug
- ❑ Le interferenze sono la conseguenza di eventi di thread o processi diversi che *si mescolano nel tempo* in modo indesiderato o imprevisto; occorre considerare il modo con cui gli eventi si verificano sulla macchina fisica

Esempio di interferenza



T_i

...
contatore := contatore + 1
...

T_j

...
contatore := contatore + 1
...

LOAD
INCR
STORE

A,contatore
A
contatore,A

in assembly

t0:	LOAD	A,contatore	(T _i)
t1:	LOAD	A,contatore	(T _j)
t2:	INCR	A	(T _j)
t3:	STORE	contatore,A	(T _j)
t4:	INCR	A	(T _i)
t5	STORE	contatore,A	(T _i)

possibile sequenza
di esecuzione



Esempio di interferenza

- Processo P: incrementa una variabile v
- Processo Q: stampa il valore di v e lo azzerava

P
 $v := v+1;$
...

Q
print $v;$
 $v := 0;$

- Le istruzioni di P e Q possono mescolarsi arbitrariamente e dar luogo a diverse possibili sequenze di esecuzione:

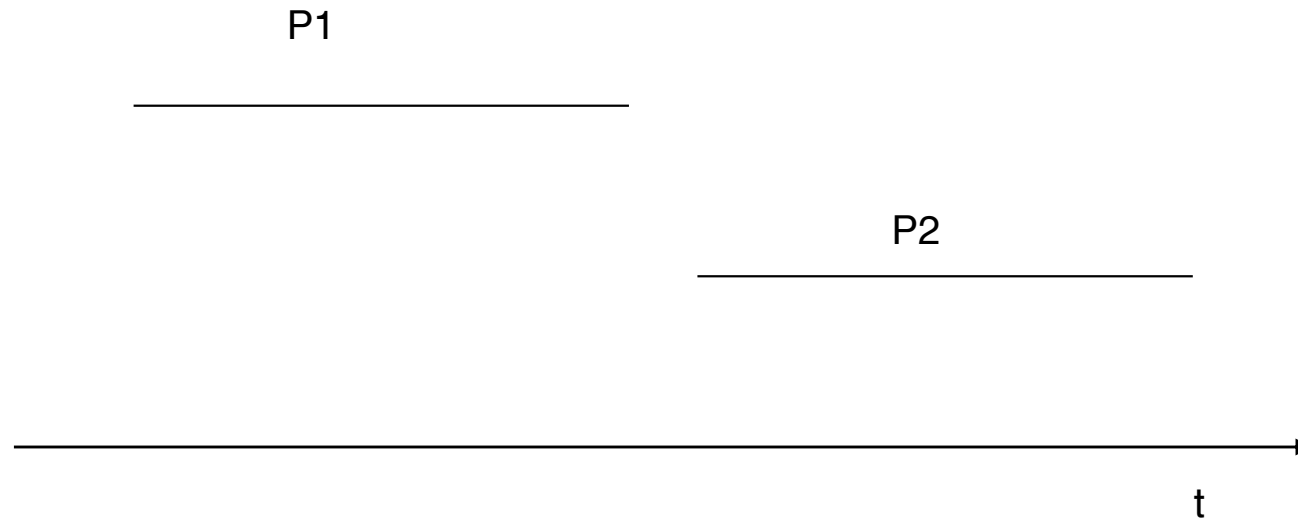
$v := v+1; (P)$	<i>print</i> $v; (Q)$	<i>print</i> $v; (Q)$
<i>print</i> $v; (Q)$	$v := 0; (Q)$	$v := v+1; (P)$
$v := 0; (Q)$	$v := v+1; (P)$	$v := 0; (Q)$
corretta	corretta	sbagliata

- In entrambi gli esempi il problema è la necessità di *serializzare* sequenze di eventi

Mutua Esclusione



- Si ha mutua esclusione quando non più di un processo alla volta può accedere a un insieme di *variabili comuni*



- *Nessun vincolo* è imposto *sull'ordine* con il quale le operazioni sulle variabili comuni sono eseguite



Esempio di mutua esclusione

- Una pagina di storia: P1 e P2 utilizzano una stessa *telescrivente (TTY)*
- La telescrivente deve essere assegnata ad un solo processo alla volta per tutta la durata del suo uso

- Ipotesi di soluzione - P1 e P2 eseguono:

```
...  
repeat until libera;    // Richiesta  
libera := false;        //  
<uso TTY>  
libera := true;         // Rilascio  
...
```

Valore iniziale: libera := *true*;

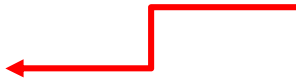




Esempio di mutua esclusione

- | | |
|---|---|
| <u>Richiesta</u>
<i>repeat until libera;</i>
<i>libera := false;</i>
... | <u>Rilascio</u>
<i>libera := true;</i>
...
(v.i. <i>libera := true</i>) |
|---|---|

- Possibile sequenza errata di esecuzione:

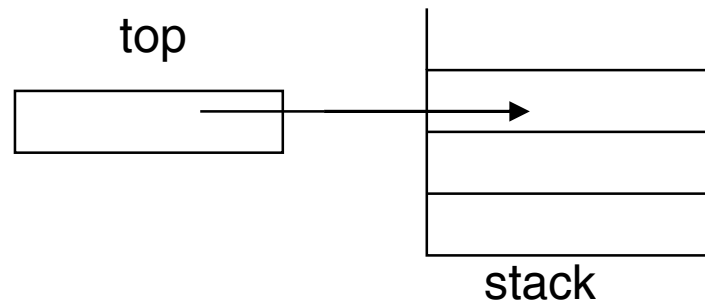
<i>t0: repeat until libera; (P1)</i> <i>t1: repeat until libera; (P2)</i> <i>t2: libera := false; (P1)</i> <i>t3: libera := false; (P2)</i>	 thread switch
--	--

- La telescrivente risulta *assegnata ad entrambi i processi*



Esempio di mutua esclusione

- Due processi hanno accesso ad una struttura dati organizzata a *pila* (stack) per depositare e prelevare messaggi:



- Inserimento(y)

...

$\text{top} := \text{top} + 1;$

$\text{stack}[\text{top}] := y;$

...

- Prelievo(x)

...

$x := \text{stack}[\text{top}];$

$\text{top} := \text{top} - 1;$

...



Esempio di mutua esclusione

- ❑ Un'*esecuzione contemporanea* delle due procedure *può* portare ad un *uso scorretto* della risorsa

- ❑ Esempio - P1 inserisce e P2 preleva:
 - t0: $\text{top} := \text{top} + 1;$ (P1)
 - t1: $x := \text{stack}[\text{top}];$ (P2)
 - t2: $\text{top} := \text{top} - 1;$ (P2)
 - t3: $\text{stack}[\text{top}] := y;$ (P1)

- ❑ Lo stesso problema si ha con riferimento all'*esecuzione contemporanea di una qualunque delle due operazioni* da parte dei due processi

Sezione critica



- ❑ La sequenza di istruzioni con le quali un processo accede e modifica un *insieme di variabili comuni* prende il nome di ***sezione critica***
- ❑ Ad un insieme di variabili comuni possono essere associate *una sola* sezione critica (usata da tutti i processi) o *più* sezioni critiche (*classe* di sezioni critiche)
- ❑ La *regola di mutua esclusione* stabilisce che:
"sezioni critiche appartenenti alla stessa classe devono escludersi mutuamente nel tempo"
ovvero:
"una sola sezione critica di una classe può essere in esecuzione ad ogni istante"

Soluzione al problema della mutua esclusione



- ❑ *Tempificazione dell'esecuzione* dei singoli processi da parte del programmatore (rischio di *errori "time dependent"*)
- ❑ *Inibizione delle interruzioni* del processore sul quale sono eseguite le sezioni critiche durante l'esecuzione di ciascuna di esse (*soluzione parziale ed inefficiente*)
- ❑ *Strumenti di sincronizzazione*

Schema generale



- Ad ogni *classe* di sezioni critiche viene associato un *indicatore*:
 libero → nessuna sezione critica in esecuzione
 occupato → una sezione critica in esecuzione
- Ogni processo che vuole utilizzare una sezione critica della classe esegue una precisa sequenza di azioni (protocollo) che inserisce l'accesso alla sezione critica tra un prologo e un epilogo:

prologo
<sezione critica>
epilogo

Schema generale: protocollo di accesso



Prologo {

- a) analizza lo stato dell'indicatore: se è occupato il processo rimane in attesa della sua liberazione; diversamente, esegue b)
- b) modifica lo stato dell'indicatore (indicatore := occupato)
}
- c) <esecuzione della sezione critica>

Epilogo {

- d) modifica lo stato dell'indicatore (indicatore := libero)
}



Mutua esclusione: prima soluzione

Prologo: disabilitazione delle interruzioni

Epilogo: abilitazione delle interruzioni

Processo P

...

disabilitazione interruzioni

<A>

abilitazione interruzioni

...

Processo Q

...

disabilitazione interruzioni

abilitazione interruzioni

...

- ❑ Nota: l'accesso alle sezioni critiche (A, B, ...) può essere richiesto in punti arbitrari del codice



Mutua esclusione: prima soluzione

- ❑ Inconvenienti:
 1. Soluzione parziale: applicabile solo in sistemi uniprocessore
 2. La soluzione non si limita ad escludere mutuamente sezioni critiche della stessa classe, ma elimina ogni possibilità di parallelismo
 3. La soluzione rende insensibile il sistema ad ogni stimolo esterno per tutta la durata di una qualunque sezione critica

- ❑ Requisito n. 1 per una soluzione accettabile:

"Le sezioni critiche devono essere eseguite con *interruzioni abilitate*"



Mutua esclusione: seconda soluzione

var LIBERO: boolean;

LIBERO := true; {inizializzazione}

processo P

...

repeat until LIBERO;
LIBERO := false;

<A>

LIBERO := true;

...

prologo

epilogo

processo Q

...

repeat until LIBERO;
LIBERO := false;

LIBERO := true;

...



Mutua esclusione: seconda soluzione

- ❑ La soluzione soddisfa il requisito n. 1
- ❑ Inconvenienti:
 - A seconda dei rapporti di velocità dei due processi, questi possono essere abilitati ad eseguire contemporaneamente le sezioni critiche <A> e
- ❑ Requisito n. 2:
 - "Le sezioni critiche di una stessa classe devono essere eseguite in forma *mutuamente esclusiva*"

Mutua esclusione: terza soluzione



```
var TURNO: 1..2;  
TURNO := 1;           {inizializzazione}
```

processo P:

```
    ...  
repeat until TURNO=1;  
    <A>  
    TURNO := 2;  
    ...
```

processo Q:

```
    ...  
repeat until TURNO = 2;  
    <B>  
    TURNO := 1;  
    ...
```



Mutua esclusione: terza soluzione

- ❑ La soluzione soddisfa ai requisiti n. 1 e 2
- ❑ Inconvenienti:
 - i processi sono trattati *in modo non uniforme*, dando priorità ad uno di essi
 - le esecuzione delle sezioni critiche *si alternano*: se un processo si blocca prima della sezione critica, l'altro non può entrare
- ❑ Requisito n. 3:
 - "Se un processo si *blocca all'esterno di una sezione critica*, ciò non deve influenzare gli altri processi"

Mutua esclusione: quarta soluzione



```
var LIBERO1, LIBERO2: boolean;  
LIBERO1 := LIBERO2 := false;           {inizializzazione}
```

processo P:

```
...  
LIBERO1 := true;  
repeat until not LIBERO2;  
    <A>  
LIBERO1 := false;  
...
```

processo Q:

```
...  
LIBERO2 := true;  
repeat until not LIBERO1;  
    <B>  
LIBERO2 := false;  
...
```



Mutua esclusione: quarta soluzione

- ❑ La soluzione soddisfa ai requisiti n. 1 e 2:
 - P entra nella sez. critica se LIBERO1 = true e LIBERO2 = false
 - Q entra nella sez. critica se LIBERO1 = false e LIBERO2 = true
- ❑ La soluzione soddisfa al requisito n. 3:
 - se P si blocca fuori dalla sez. critica, LIBERO1 rimane false e Q può entrare
- ❑ Inconvenienti:
 - a seconda dei rapporti di velocità dei due processi si può avere una condizione di *stallo* (o *deadlock*)
 - in particolare: P e Q contemporaneamente su repeat (*)
- ❑ Requisito n. 4:
 - "La soluzione al problema della mutua esclusione deve essere *esente da condizioni di stallo*"

Mutua esclusione: ulteriori requisiti



- ❑ Per motivi di efficienza nell'uso delle risorse è opportuno che un processo *rilasci il controllo della CPU quando deve rimanere in attesa*
- ❑ Requisito n. 5:
 - "La soluzione non deve presentare fenomeni di *attesa attiva (busy waiting)*"
- ❑ La richiesta di accesso di un processo deve venire *prima o poi soddisfatta*, indipendentemente dalle priorità e velocità relative dei processi
- ❑ Requisito n. 6:
 - "La soluzione non deve *disabilitare per sempre* uno o più processi (*starvation*)"

Coordinazione distribuita



Algoritmo di Dekker



```
var LIBERO1, LIBERO2: boolean;      TURNO: 1..2;  
LIBERO1 := LIBERO2 := false; TURNO := 1; {inizializz.}
```

processo P:

```
...  
LIBERO1 := true;  
while LIBERO2 do  
  if TURNO = 2 then  
    begin  
      LIBERO1 := false;  
      repeat until TURNO = 1;  
      LIBERO1 := true;  
    end;
```

<A>

```
TURNO := 2;  
LIBERO1 := false;
```

...

processo Q:

```
...  
LIBERO2 := true;  
while LIBERO1 do  
  if TURNO = 1 then  
    begin  
      LIBERO2 := false;  
      repeat until TURNO = 2;  
      LIBERO2 := true;  
    end;
```



```
TURNO := 1;  
LIBERO2 := false;
```

...

Algoritmo di Dekker



- Proprietà:
- R2: I due processi non possono essere simultaneamente nella sezione critica:
 - solo Q modifica LIBERO2, solo P modifica LIBERO1
 - P entra nella sez. critica solo se LIBERO2 = false
 - P verifica LIBERO2 dopo aver posto LIBERO1 := true
- R3: Se un solo processo richiede di entrare nella sez. critica il suo accesso è sempre garantito (condizione *while* non soddisfatta)
- R4: Nel caso di tentativi di accesso simultanei (LIBERO1 e LIBERO2 entrambi true) la variabile TURNO consente di arbitrare l'accesso dei processi. Non c'è deadlock.
- R6: Non c'è starvation. Nel caso peggiore (processo P interrotto subito dopo il *repeat until*) il processo Q può guadagnare l'accesso solo per una ulteriore volta prima che P entri

Algoritmo di Dekker



- ❑ Inconvenienti:
- ❑ busy waiting !
- ❑ L'estensione al caso di N processi non è banale: diventa rilevante il problema della starvation e l'esigenza di minimizzare l'attesa nel caso peggiore.

Algoritmo di Peterson



```
var LIBERO1, LIBERO2: boolean;  TURNO: 1..2;  
LIBERO1 := LIBERO2 := false;    {inizializzazione}
```

processo P:

...

```
LIBERO1 := true;  
TURNO := 2;  
while LIBERO2  
    and TURNO = 2 do;
```

<A>

```
LIBERO1 := false;
```

...

processo Q:

...

```
LIBERO2 := true;  
TURNO := 1;  
while LIBERO1  
    and TURNO = 1 do;
```



```
LIBERO2 := false;
```

...

Algoritmo di Peterson



- ❑ R2 - mutua esclusione:
- ❑ I due processi non possono essere simultaneamente nella sezione critica:
 - solo Q modifica LIBERO2, solo P modifica LIBERO1,
 - P entra nella sez. critica solo se $\text{LIBERO1} = \text{true}$ *and* ($\text{LIBERO2} = \text{false}$ *or* $\text{TURN0} = 1$),
 - Q entra nella sez. critica solo se $\text{LIBERO2} = \text{true}$ *and* ($\text{LIBERO1} = \text{false}$ *or* $\text{TURN0} = 2$),
 - P manipola la variabile TURN0 solo prima di entrare nel *while*, cioè prima di testare LIBERO2. Analogamente Q.
- ❑ R3 - indipendenza delle sezioni critiche:
- ❑ Se un solo processo richiede di entrare nella sez. critica il suo accesso è sempre garantito (condizione *while* non soddisfatta)



Algoritmo di Peterson

- ❑ R4 - assenza di deadlock:
- ❑ Le possibili condizioni sono:
 - LIBERO1 = false, LIBERO2 = false: processi fuori dalla sez. critica,
 - LIBERO1 = true, LIBERO2 = false: P nella sezione critica,
 - LIBERO1 = true, LIBERO2 = true, TURNO = 1: entra P,
- ❑ e dualmente. Per qualunque condizione di stato non si ha deadlock.
- ❑ R6 - assenza di starvation:
- ❑ Il processo in attesa P_i potrebbe logicamente procedere non appena la sezione critica viene liberata da P_j ($LIBERO_J = false$). Un eventuale nuovo tentativo di accesso da parte del processo P_j prima della esecuzione di P_i rimane in coda poiché P_j pone $TURNO := i$.

Algoritmo del Fornaio (Lamport, 1974)



- ❑ Algoritmo di mutua esclusione per N processi (con busy waiting)
- ❑ Modella il problema della mutua esclusione come il servizio di clienti in un negozio
 - Quando entra nel negozio (richiesta di accesso alla sezione critica) ciascun cliente (processo) riceve *un numero*. Viene servito il cliente che possiede il numero più basso.
 - In questo caso però più clienti possono ricevere lo *stesso numero*!
- ❑ Variabili *condivise* tra gli N processi:

```
var CHOOSING: array [0 .. N - 1] of boolean initial false;  
    NUMBER:   array [0 .. N - 1] of integer initial 0;
```

Algoritmo del Fornaio



repeat

...

CHOOSING[i] := true;

NUMBER[i] := 1+max(NUMBER[0], ..., NUMBER[N-1]);

CHOOSING[i] := false;

for j := 0 *to* N-1

do begin

while CHOOSING[j] *do*;

while NUMBER[j] <> 0

and (NUMBER[j],j) < (NUMBER[i],i) *do*;

end;

<A>

NUMBER[i] := 0;

...

forever;

*Eseguito da ciascun processo P_i che
partecipa alla sezione critica*

Algoritmo del Fornaio



- Il prologo è costituito da due fasi: la *acquisizione del numero d'ordine* (intervallo in cui CHOOSING = true) e la *attesa del turno di servizio* (CHOOSING = false)
- E' possibile che più clienti ricevano *lo stesso numero* (per interleaving nella fase di acquisizione). In tal caso viene seguito *l'ordine alfabetico*. Poiché gli identificatori dei processi sono univoci, la scelta è *deterministica*.
- Nota: $(a,b) < (c,d)$ equivale a $a < c$ or $(a = c \text{ and } b < d)$

Algoritmo del Fornaio



- Proprietà:
- R1: Interrupt abilitati
- R2: Mutua esclusione
 - Se due clienti P_i e P_k si trovano nel negozio e P_i è entrato nel negozio prima che P_k acquisisse il numero:
 $\text{NUMBER}[i] < \text{NUMBER}[k]$
 - Se P_i è nella sezione critica e P_k è nel negozio:
 $(\text{NUMBER}[i], i) < (\text{NUMBER}[k], k)$
 - Ne consegue che al più un processo può trovarsi nella sezione critica
- R3: L'indipendenza degli accessi alle sezioni critiche della classe è ovviamente soddisfatta

Algoritmo del Fornaio



- Proprietà /2/:
- R4: Assenza di condizioni di stallo
 - L'ordinamento totale (NUMBER[i],i) assicura che uno (ed uno solo) dei processi in attesa può entrare nella sezione critica quando essa si libera
- R6: Assenza di starvation
 - I processi vengono serviti in ordine *First-Come, First-Served* (importante nel caso di N processi); il *numero massimo di turni di attesa* di un processo è N-1



Algoritmo del Fornaio

- ❑ L'algoritmo del fornaio è stato concepito per essere utilizzato anche in un *ambiente decentralizzato*, in cui ogni processo dispone di una memoria locale ma può accedere *in lettura* alle memorie degli altri processi. In tal caso gli array CHOOSING e NUMBER sono distribuiti sulle diverse memorie.
- ❑ Nel caso di realizzazione decentrata, l'eventuale guasto di un nodo *non blocca gli altri processi*, nell'ipotesi che le letture degli altri processi dalla memoria guasta restituiscano il valore 0 (proprietà di *graceful degradation*).
- ❑ Inconvenienti: busy waiting
- ❑ Numero fisso e noto di processi / thread

Algoritmi del Fornaio, Dekker, Peterson



- ❑ Inconvenienti:
 - Busy waiting
 - Numero fisso e noto di processi/thread
 - I processi devono conoscersi e coordinarsi reciprocamente

- ❑ Sono esempi di *algoritmi di coordinazione distribuita*

- ❑ Non forniscono una soluzione sistemistica al problema della mutua esclusione